

Developing Zynq Software with Xilinx SDK

Lab 9

Dual Processor Software Development



August 2016
Version 09

Lab 9 Overview

The Zynq®-7000 All Programmable SoC contains two Cortex®-A9 processors which share common memory and peripherals. These processor cores can be configured to concurrently run independent software stacks or executable code. Asymmetric multiprocessing (AMP) is a mechanism that allows both processors to run their own operating systems or bare-metal applications with the possibility of loosely coupling those applications via shared resources.

This lab describes a method of starting up both processors, each running its own bare-metal software application, and allowing each processor to communicate with the other through shared memory.

Lab 9 Objectives

When you have completed Lab 9, you will know how to do the following:

- Create a bare-metal application with its own standalone environment targeted for the first processor core CPU0
- Create a second bare-metal application with its own standalone environment targeted for the second processor core CPU1
- Generate a bootable solution which loads each application onto the respective processor core
- Debug the execution of each application from the respective processor core

Experiment 1: Creating Bare-Metal Application Project for CPU0

This experiment shows how to create the application ELF that runs on CPU0 after the FSBL copies the application ELFs to DDR memory.

Experiment 1 General Instruction:

Launch Xilinx Software Development Kit (SDK) and open the SDK workspace found in the following folder:

C:\Speedway\ZynqSW\2016_2\SDK_Workspace

Create a new empty application project and import the CPU0 application found in the **Support_documents** folder.

Experiment 1 Step-by-Step Instructions:

1. Launch Xilinx Software Development Kit (SDK) if not already open. **Start → All Programs → Xilinx Design Tools → SDK 2016.2 → Xilinx SDK 2016.2.**



Figure 1 – The SDK Application Icon

2. Set or switch the workspace to the following folder and then click the **OK** button:

C:\Speedway\ZynqSW\2016_2\SDK_Workspace

3. Create a new SDK software application project by selecting the **File→New→Application Project** menu item.
4. In the **New Project** wizard, change the **Project name** field to **app_cpu0** and leave the other settings to their default values. Click the **Next** button to continue.

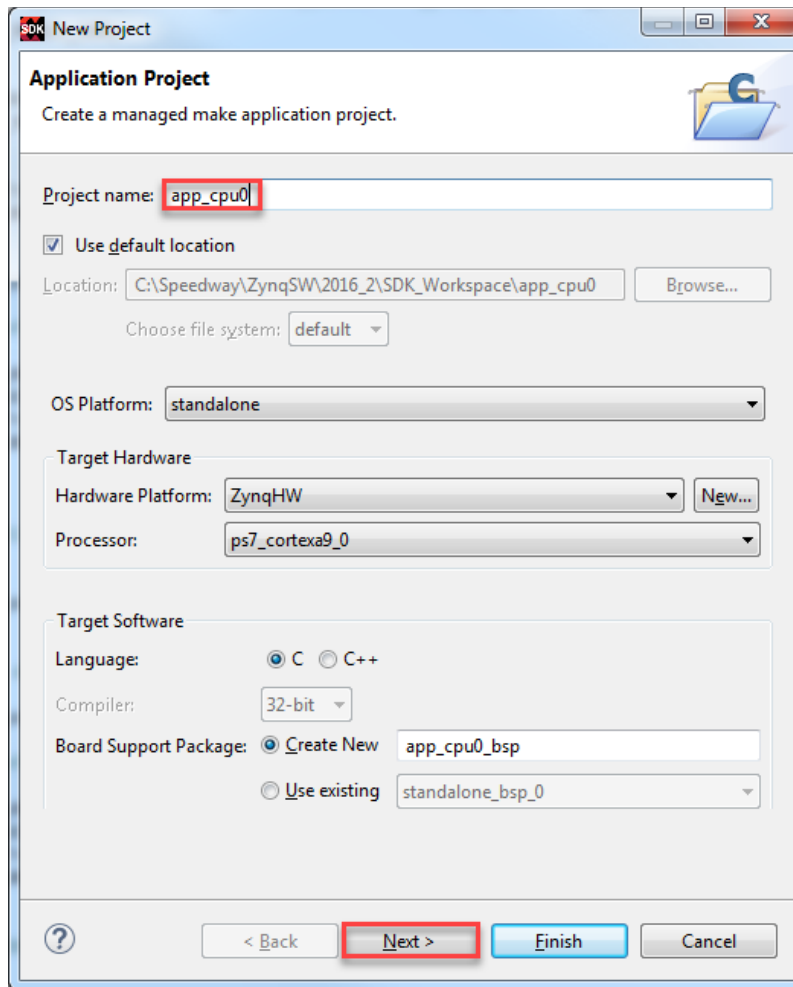


Figure 2 – Creating the CPU0 Application

5. Select the **Empty Application** project template and click the **Finish** button to complete the new project creation process using the **Empty Application** project template.
6. The application project is created along with the corresponding BSP **app_cpu0_bsp** and each of the projects will be compiled automatically.

In the **Project Explorer** tab, expand **app_cpu0** and right-click on the **src** folder.

Click on the **Import** option in the pop up menu.

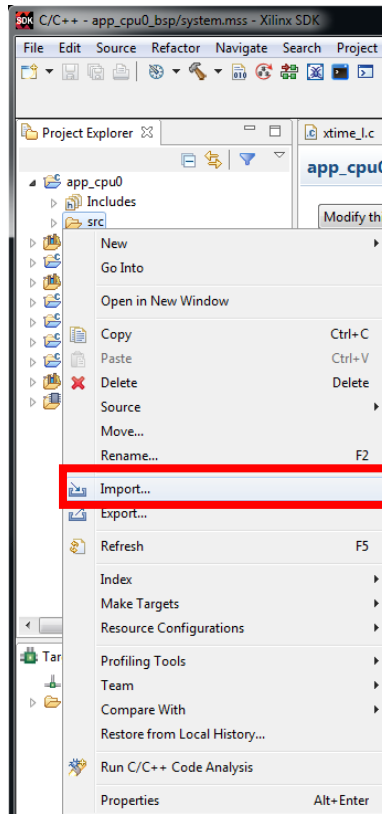


Figure 3 – Import CPU0 Application Code

7. In the **Import** window, expand the **General** item, select the **File System** option, and click the **Next** button.

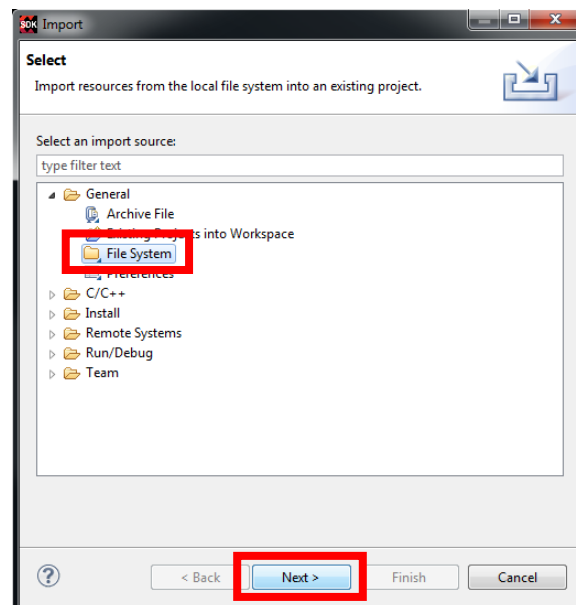


Figure 4 – Importing from a File System

- Click on the **Browse** button and select the following folder which contains the application code that we wish to run on CPU0:

C:\Speedway\ZynqSW\2016_2\Support_documents\app_cpu0

After this folder is selected within the **Browse** dialog, click the **OK** button to search the folder for files to import into the application project.

Select only the **app_cpu0.c** source file by checking the box next to the file name and then click the **Finish** button to complete the **Import** operation.

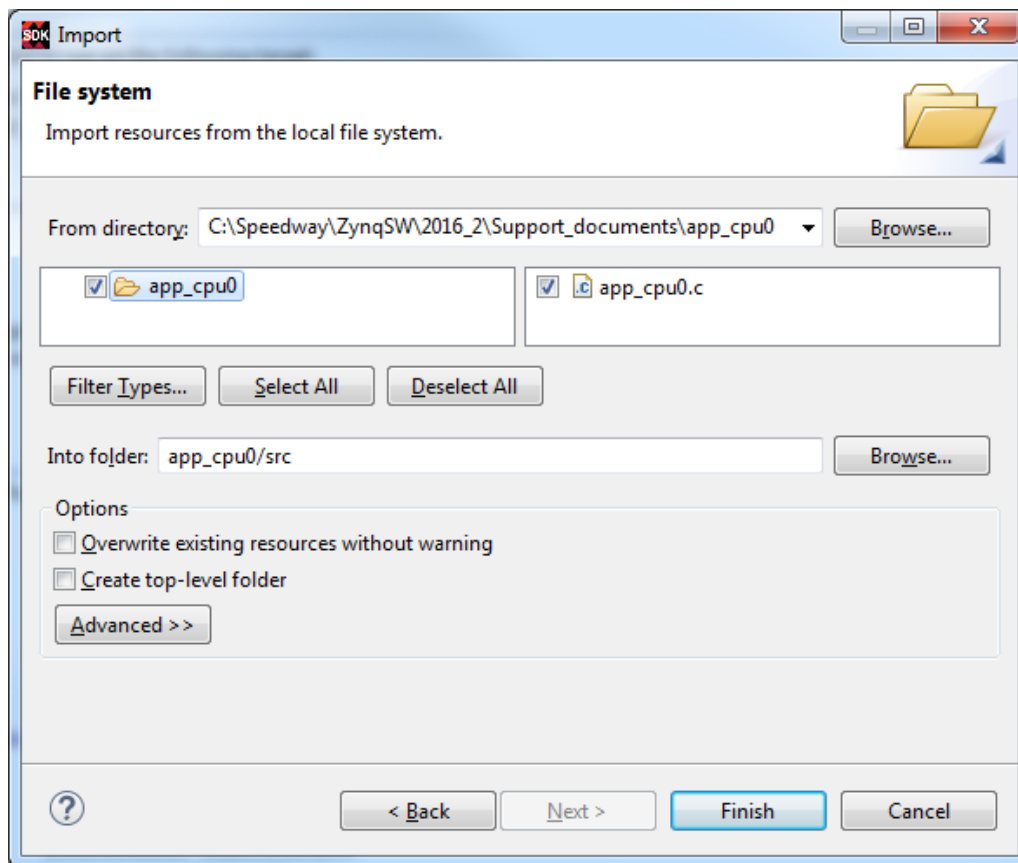


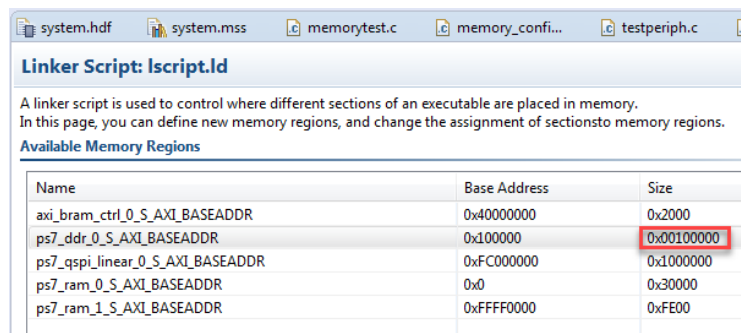
Figure 5 – Selecting Application Source Files

9. Open the existing **app_cpu0** project Linker Script **lscript.ld** file. This project was generated automatically when the empty application project was created. The entire memory space from 0x00100000 to 0x3FFFFFFF is assigned to this application which is not only more than is needed for our AMP application, but we also need some separate memory space for the CPU1 application to execute out of later.

Also, if you were to look at the **app_cpu0.c** source code you will find that the CPU1 application memory space is expected to start at 0x00200000 and there is another shared memory region which starts at address 0x03000000 so unless source code modifications are made, we will want to avoid using those memory areas when loading the CPU0 application code.

Modify the DDR memory region definition within **lscript.ld** by setting the size of the region to **0x00100000** such that **app_cpu0.elf** occupies the memory space from 0x00100000 to 0x001FFFFFFF instead of the entire DDR memory space.

Save the changes to **lscript.ld** to rebuild the **app_cpu0** project with the new memory space definition.

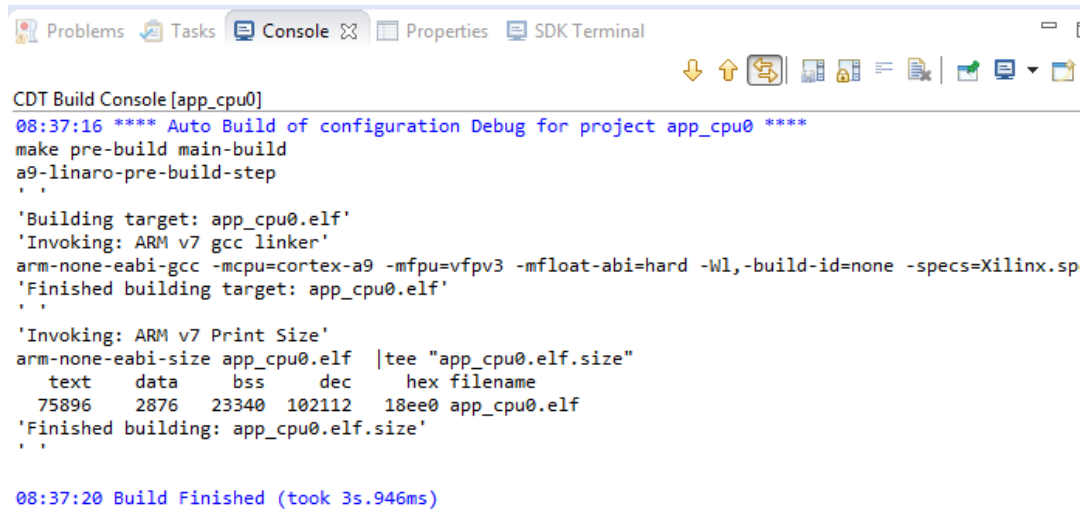


The screenshot shows the 'Linker Script: lscript.ld' editor. It includes a table of 'Available Memory Regions'. The row for 'ps7_ddr_0_S_AXI_BASEADDR' has its 'Size' column value '0x00100000' highlighted with a red box. Other rows include 'axi_bram_ctrl_0_S_AXI_BASEADDR' (Size: 0x2000), 'ps7_qspi_linear_0_S_AXI_BASEADDR' (Size: 0x1000000), 'ps7_ram_0_S_AXI_BASEADDR' (Size: 0x30000), and 'ps7_ram_1_S_AXI_BASEADDR' (Size: 0xFE00).

Name	Base Address	Size
axi_bram_ctrl_0_S_AXI_BASEADDR	0x40000000	0x2000
ps7_ddr_0_S_AXI_BASEADDR	0x1000000	0x00100000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x0	0x30000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0xFE00

Figure 6 – Modifying lscript.ld

10. The SDK Console panel shows the results of the build. Make sure that the application is built without errors.



```
CDT Build Console [app_cpu0]
08:37:16 **** Auto Build of configuration Debug for project app_cpu0 ****
make pre-build main-build
a9-linaro-pre-build-step
',
'Building target: app_cpu0.elf'
'Invoking: ARM v7 gcc linker'
arm-none-eabi-gcc -mcpu=cortex-a9 -mfpv=vfpv3 -mfloat-abi=hard -Wl,-build-id=none -specs=Xilinx.sp
'Finished building target: app_cpu0.elf'
',
'Invoking: ARM v7 Print Size'
arm-none-eabi-size app_cpu0.elf |tee "app_cpu0.elf.size"
  text    data    bss     dec     hex filename
 75896    2876   23340  102112  18ee0 app_cpu0.elf
'Finished building: app_cpu0.elf.size'
',
08:37:20 Build Finished (took 3s.946ms)
```

Figure 7 – Application Build Console Window

11. After SDK finishes compiling the new application code, the ELF is available in the following location:

C:\Speedway\ZynqSW\2016_2\SDK_Workspace\app_cpu0\Debug\app_cpu0.elf

Questions:

Answer the following questions:

- What is the difference between a normal standalone application running on CPU0 and a standalone application running on CPU1?

Experiment 2: Creating Bare-Metal Application Project for CPU1

This experiment shows how to create the application ELF that runs on CPU1 after the FSBL copies the application ELFs to DDR memory.

This step is slightly different than the previous experiment in creating the application for CPU0 because the CPU1 application uses a customized BSP which is adapted from Xilinx XAPP1079 – *Simple AMP Bare-Metal System Running on Both Cortex-A9 Processors*. An important aspect of this design is that it prevents CPU1 from re-initializing shared resources. In this example application, CPU1 does not use L2 cache which is a shared resource and CPU0 “owns” this resource. If CPU1 were to use L2 cache, L2 cache flushes and invalidates would need to be requested from CPU0 and CPU0 would then exercise the action. It is beyond the scope of this Exercise to include a communication channel that enables CPU1 to request L2 cache interactions.

SDK is used to create the BSP using the customized standalone BSP from the repository which is included with the lab files.

Experiment 2 General Instruction:

Create a new Board Support Package to use an Standalone BSP that utilized ps7_cortexa9_1

Create a new empty application project and import the CPU1 application found in the **Support_documents** folder.

Experiment 2 Step-by-Step Instructions:

1. Create a new SDK Board Support Package by selecting the **File→New→ Board Support Package** menu item.

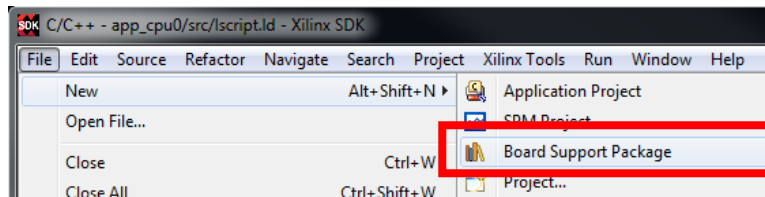


Figure 8 – Creating a New Board Support Package

2. In the **New Board Support Package Project** window, change the **Project name** field to **app_cpu1_bsp** to match the CPU1 application which will be created in later steps.

Use the drop down menu to change the **CPU** setting to the **ps7_cortexa9_1** option.

Keep the **Board Support Package OS** standalone.

Click the **Finish** button to create the new Standalone BSP.

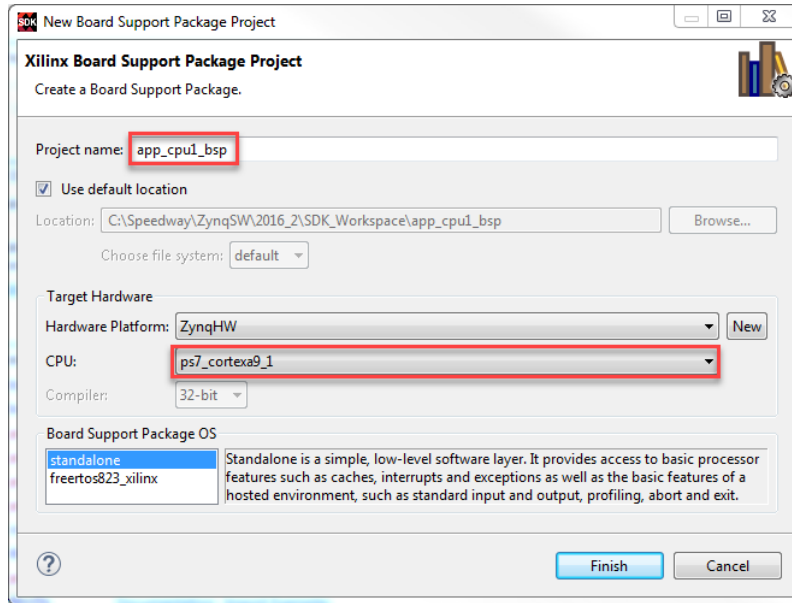


Figure 9 – New Standalone Board Support Package Project

3. In the **Board Support Package Settings** window for the newly created **app_cpu1_bsp** project, navigate to the **ps7_cortexa9_1** driver configuration under the **Overview→Drivers** item listing.

Add the following compiler flag string to the existing **extra_compiler_flags** property:

-DUSE_AMP=1

Make sure to place a **space** in front of the **-DUSE_AMP=1** compiler flag

This compiler flag definition enables some code sections which prevent the re-initialization of shared CPU resources such as the SCU and L2 cache.

Click on the **OK** button to accept the new driver settings for the Board Support Package.

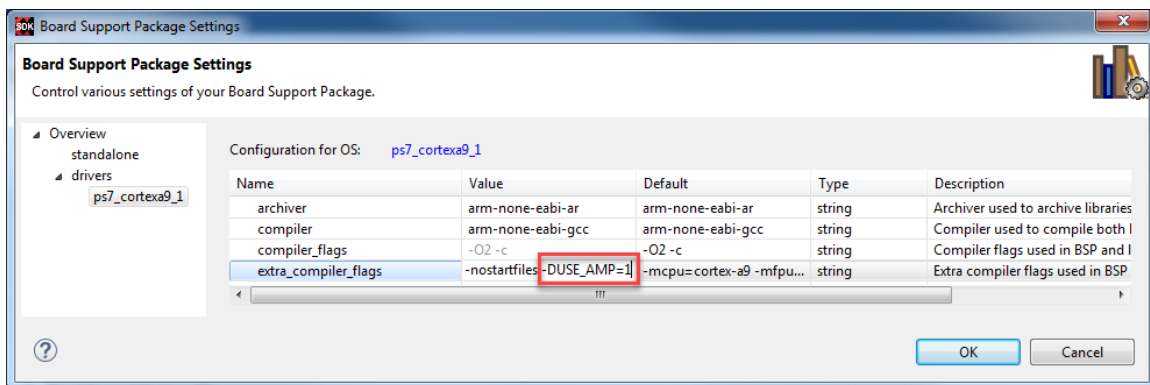


Figure 80 – Modifying Settings for New Board Support Package

4. Create a new SDK software application project by selecting the **File→New→Application Project** menu item.
5. In the **New Project** wizard, change the **Project name** field to **app_cpu1** to match the CPU1 BSP which was created in Step 4.

Use the drop down menu to change the **Processor** setting to the **ps7_cortexa9_1** option.

You can leave the **OS Platform** setting as the **standalone** option.

Change the **Board Support Package** to the **Use existing** option and make sure the BSP selected is **app_cpu1_bsp**.

Click the **Next** button to continue.

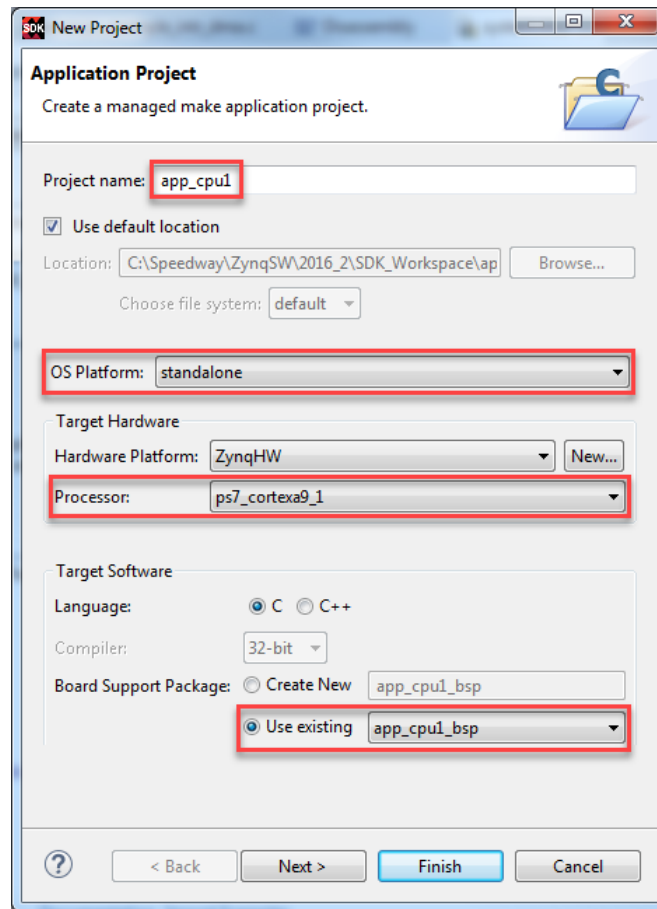


Figure 91 – Creating the CPU1 Application

6. Select the **Empty Application** project template and click the **Finish** button to complete the new project creation process using the **Empty Application** project template.

7. The empty **app_cpu1** application project is created.

In the **Project Explorer** tab, expand **app_cpu1** and right-click on the **src** folder.

Click on the **Import** option in the pop up menu. In the **Import** window, expand the **General** item, select the **File System** option, and click the **Next** button.

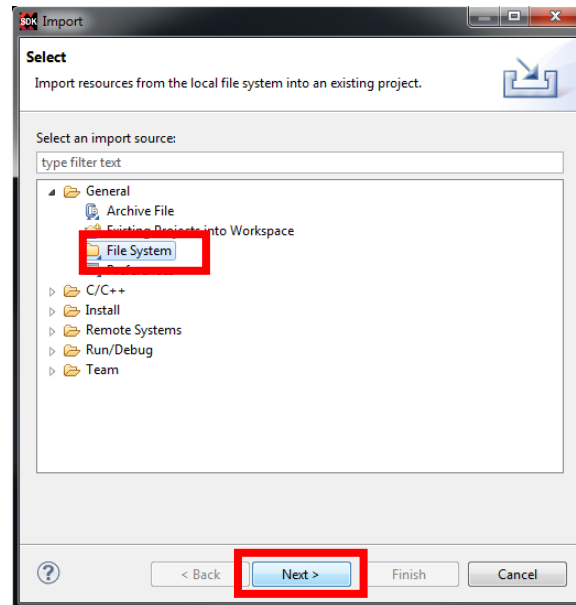


Figure 102 – Importing from a File System

8. Click on the **Browse** button and select the following folder which contains the application code that we wish to run on CPU1:

C:\Speedway\ZynqSW\2016_2\Support_documents\app_cpu1

After this folder is selected within the **Browse** dialog, click the **OK** button to search the folder for files to import into the application project.

Select only the **app_cpu1.c** source file by checking the box next to the file name and then click the **Finish** button to complete the **Import** operation.

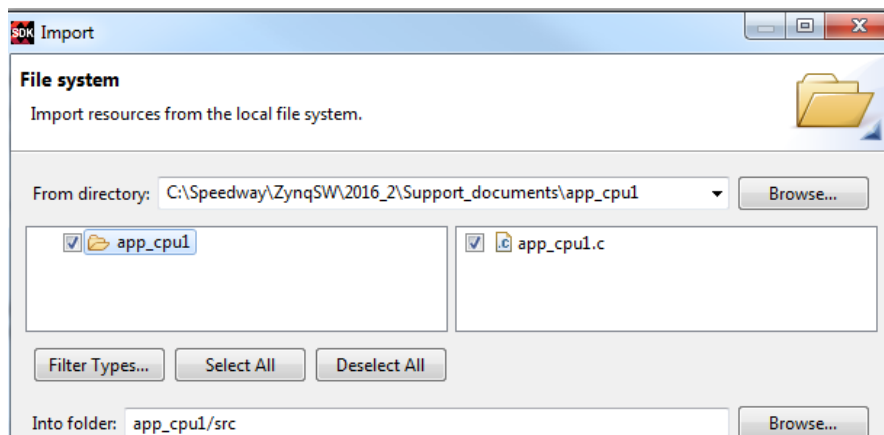


Figure 113 – Selecting Application Source Files

9. Open the existing **app_cpu1** project Linker Script **lscript.ld** file. This project was generated automatically when the empty application project was created. The entire memory space from 0x00100000 to 0x3FFFFFFF is assigned to this application which is not only more than is needed for our AMP application, but we also need to reserve a separate memory space for the CPU0 application to execute out of also.

Also, if you were to look at the **app_cpu0.c** source code you will find that the CPU1 application memory space is expected to start at 0x00200000 so that memory location should become the start of the region for **app_cpu1**. There is another shared memory region which starts at address 0x03000000 so unless source code modifications are made, we will want to avoid using that memory area when loading the CPU1 application code.

Modify the DDR memory region definition within **lscript.ld** by setting the base address for that region to **0x00200000** and the size of the region to **0x00100000** such that **app_cpu1.elf** occupies the memory space from 0x00200000 to 0x002FFFFFFF instead of the entire DDR memory space.

IMPORTANT NOTE: Be sure to double check that the above address modification is performed. Otherwise, when CPU1 application is loaded into memory by the FSBL, the loaded CPU0 application will be overwritten with the CPU1 application. Once Zynq boots into this condition, this will have the effect of hanging the CPU0 core execution while CPU1 will sleep waiting for an event from CPU0 which will never happen.

Save the changes to **lscript.ld** to rebuild the **app_cpu1** project with the new memory space definition.

Linker Script: Iscript.ld

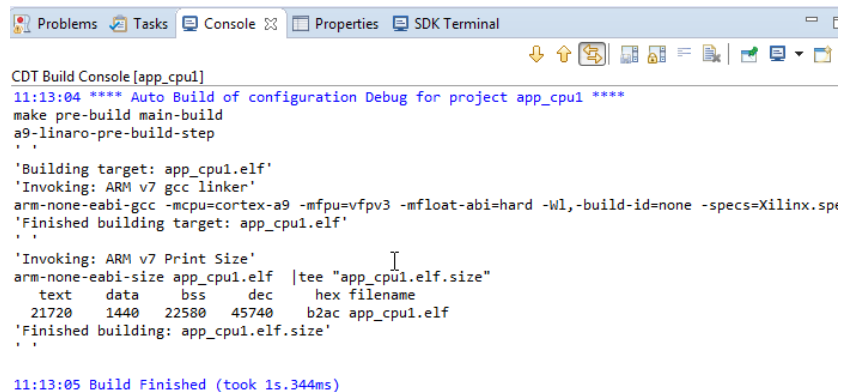
A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

Available Memory Regions

Name	Base Address	Size
axi_bram_ctrl_0_S_AXI_BASEADDR	0x40000000	0x2000
ps7_dds_0_S_AXI_BASEADDR	0x00200000	0x00100000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x0	0x30000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0xFE00

Figure 124 – Modifying Iscript.ld

10. The SDK Console panel shows the results of the build. Make sure that the application is built without errors.



```
CDT Build Console [app_cpu1]
11:13:04 **** Auto Build of configuration Debug for project app_cpu1 ****
make pre-build main-build
a9-linaro-pre-build-step
.,
'Building target: app_cpu1.elf'
'Invoking: ARM v7 gcc linker'
arm-none-eabi-gcc -mcpu=cortex-a9 -mfpv=vfpv3 -mfloat-abi=hard -Wl,-build-id=none -specs=Xilinx.spe
'Finished building target: app_cpu1.elf'
.,
'Invoking: ARM v7 Print Size'
arm-none-eabi-size app_cpu1.elf |tee "app_cpu1.elf.size"
text data bss dec hex filename
21720 1440 22580 45740 b2ac app_cpu1.elf
'Finished building: app_cpu1.elf.size'
.,
11:13:05 Build Finished (took 1s.344ms)
```

Figure 135 – Application Build Console Window

11. After SDK finishes compiling the new application code, the ELF is available in the following location:

C:\Speedway\ZynqSW\2016_2\SDK_Workspace\app_cpu1\Debug\app_cpu1.elf

Questions:

Answer the following questions:

- Why are two separate application projects required for AMP applications?

Experiment 3: Booting the AMP Applications on Zynq

This experiment shows how to boot the two AMP applications to their respective CPU cores on Zynq.

When creating a Zynq boot image which boots AMP applications to each of the CPU cores, the following items from the SDK_Workspace folder need to be added in the order shown:

FSBL - bootloader - **zynq_fsbl_0\Debug\zynq_fsbl_0.elf**

PL Bitstream - datafile – **hw_platform\Z_system_wrapper.bit**

CPU0 Application - datafile – **app_cpu0\Debug\app_cpu0.elf**

CPU1 Application - datafile – **app_cpu1\Debug\app_cpu1.elf**

Each of these files will be built into a Zynq boot image file which will be copied to the SD card and renamed to the **boot.bin** filename. The naming of this file is important since this is the filename that the Zynq BootROM searches for when the mode pins have been configured to boot from the SD card.

The Zynq BootROM then opens the **boot.bin** file and searches for the block of data that has been flagged as the **bootloader** partition. The BootROM loads this FSBL application code into OCM and starts running it.

In turn, the FSBL loads the PL bitstream file, the CPU0 ELF, and the CPU1 ELF from their respective boot partitions. At this point, the FSBL that is running on CPU0 jumps to the execution address of the first application ELF that was loaded after the FSBL. As CPU0 starts to run **app_cpu0.elf**, it writes the starting address of the CPU1 application (0x00200000) to OCM at 0xFFFFFFFF, then executes the assembly instruction **SEV**, which sets an event that wakes up CPU1.

Before the BootROM started running the FSBL, it wrote a very small set of CPU instructions at 0xFFFFFFFF and set the CPU1 program counter to this location, effectively “parking” the core at this high instruction memory location. This small application on CPU1 checks the contents of 0xFFFFFFFF and if it is set to 0, executes the **Wait For Event (WFE)** instruction. Every time an event occurs, CPU1 wakes up and reruns the loop where it checks the contents of 0xFFFFFFFF again for a non-zero value. As soon as a non-zero value is detected, CPU1 jumps to the address location that was read from 0xFFFFFFFF.

In the case of our application, the value stored in 0xFFFFFFFF is 0x00200000, which is the starting address of the CPU1 application as defined in the **Iscrip.ld** linker script for the **app_cpu1** application.

Experiment 3 General Instruction:

Create a Zynq Boot Image containing the applications for each of the two CPU cores.

Boot the target board to run the AMP applications.

Experiment 3 Step-by-Step Instructions:

1. In the **Project Explorer** window, select the **app_cpu0** project.
2. Open the **Create Boot Image** window by selecting the **Xilinx Tools**→**Create Boot Image** menu item.

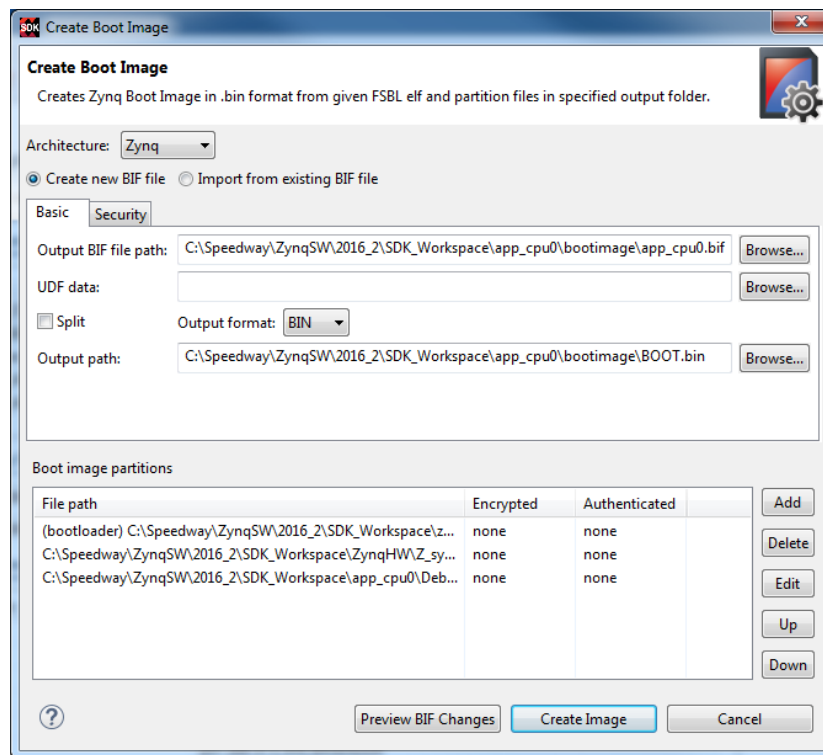


Figure 146 – Create Zynq Boot Image Tool Pre Setup

3. In the **Create Zynq Boot Image** window, use the **Browse** button next to the **BIF file path** field to create a new Boot Image Format file in the following location:

C:\Speedway\ZynqSW\2016_2\SDK_Workspace\app_cpu0\bootimage\amp_boot.bif

For the Boot image partitions list, the following item from the **SDK_Workspace** folder need to be added:

Click the **Add** button then locate the **app_cpu1.elf** and set the **Partition type** of the new partition to **datafile**. Click the **OK** button to add this new partition to the **Boot image partitions** list.

Files are added in the order shown since this is the order in which their boot image partitions will be processed by the FSBL. Once the FSBL begins processing boot image partitions, it will first encounter the PL bitstream and use the data contained within that partition to configure the PL via the PCAP interface. The next boot image partition encountered is the CPU0 application which is loaded into DDR memory space according to the definitions of the application's linker script and the application start address is stored for later use. The last boot image partition encountered is the CPU1 application executable which will be loaded into the DDR memory space defined within that application's linker script. The final FSBL step is to jump to the start address for the CPU0 application after which the FSBL will no longer execute until a POR or SRST event occurs.

Specify the output path as the following location:

C:\Speedway\ZynqSW\2016_2\SDK_Workspace\app_cpu0\bootimage\BOOT.bin

Click the **Create Image** button to place the Zynq boot image in the **Output path**.

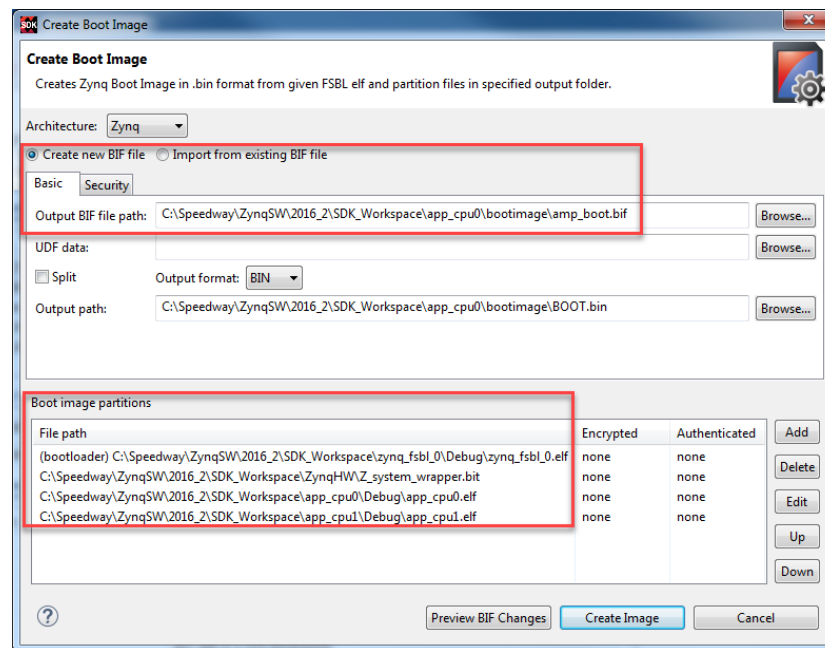


Figure 157 – Create Zynq Boot Image Tool

- Using Windows Explorer, navigate to the following folder:

C:\Speedway\ZynqSW\2016_2\SDK_Workspace\app_cpu0\bootimage\

Notice that two files have been created: **.bif** and **.bin** files. Only the **BOOT.bin** file is required to boot Zynq from the microSD/SD Card.

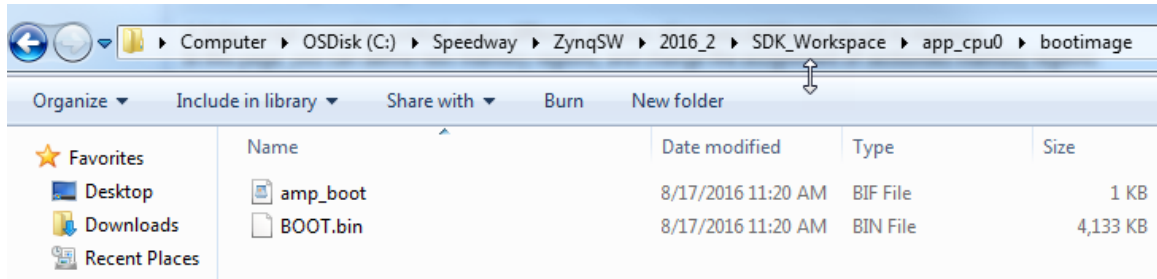


Figure 168 – bootimage Folder

5. Prepare the microSD/SD card with the Zynq Boot Image **BOOT.bin** file that SDK generated in Step 4 above.
6. To program a microSD/SD Card, insert the microSD/SD card into an adapter and plug into your PC. Copy the **BOOT.bin** image to the microSD/SD Card.
7. On the microSD/SD Card, overwrite any existing **BOOT.bin** files which may be present from earlier lab activities.
8. On the microSD/SD Card, double check that the boot image file is named **BOOT.bin** if it is named something different, then Zynq will not boot properly. **THIS STEP IS CRITICAL!** The Zynq Stage 0 bootloader looks specifically for this file. Also, the name is not case sensitive.
9. Eject the microSD/SD card from the PC.
10. *<ZedBoard & PicoZed Only>* Connect the power cable to the ZedBoard, but leave ZedBoard OFF for now.
11. Insert the microSD/SD card into the MicroZed/ZedBoard/PicoZed microSD/SD slot.
12. Set the Boot Jumpers to SD Card Boot Mode.
 - a. Zedboard and MicroZed

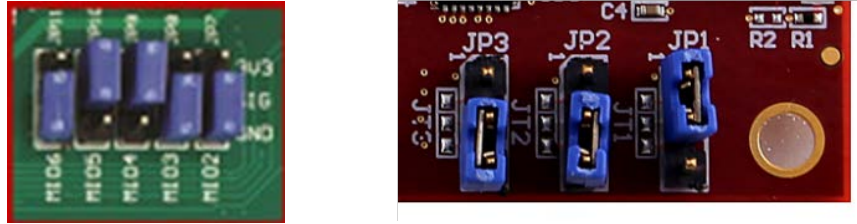


Figure 1917 – SD Card Boot Mode: ZedBoard left; MicroZed right

b. PicoZed

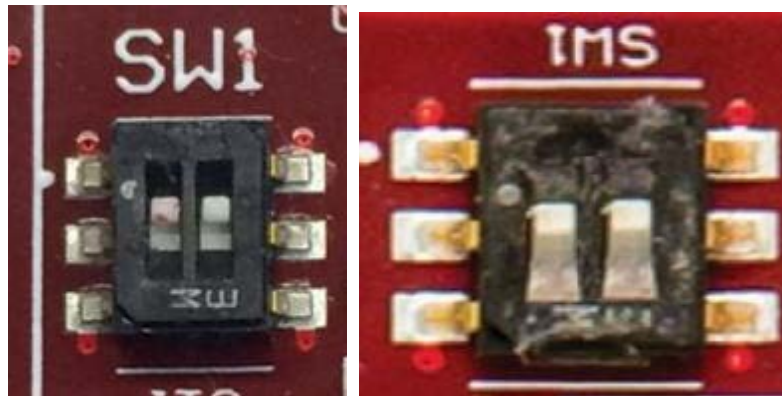


Figure 20 - PicoZed SW1 Set to SD Boot 7010/20 on the Left; 7015/30 on the Right

13. Close or disconnect the terminal that may have previously been open on your PC.
14. *<ZedBoard & PicoZed Only>* Turn power on.
15. Connect a micro-USB cable between the Windows Host machine and the USB-UART
16. Launch a terminal program (Tera Term) with the 115200/8/n/1/n settings.
17. Push the RST (SW2) button on MicroZed, CARRIER_RST_N(SW8) on PicoZed, or PS-RST (BTN7) button on ZedBoard.
18. Observe the terminal output from CPU0 and CPU1 applications.

Experiment 4: Debugging AMP Applications on Zynq

This experiment shows how to debug the AMP applications on Zynq.

Xilinx SDK can be used to connect and debug the applications running on both CPUs simultaneously. The TCF debugger provides a hardware "server" that connects to the CPU by way of the JTAG cable. Normally, SDK automatically starts TCF in the background when starting to debug an application. The same technique will be used in this experiment where TCF is automatically connected to both CPU0 and CPU1 targets. Then SDK connects to the TCF session during debug which makes both CPU core execution paths visible to the tools.

Since the FSBL was used to boot the AMP applications, there is no need to reinitialize the PS registers. Care must be taken not to reset the full PS because both CPU applications are to be debugged simultaneously.

Experiment 4 General Instruction:

Boot the target board to run the AMP applications.

Connect to the two CPU cores to debug the AMP application code using a TCF debug session.

Experiment 4 Step-by-Step Instructions:

1. Boot MicroZed/PicoZed/ZedBoard to the applications. This is the point where Experiment 3 instructions left off. The console should show a rolling stream of **print()** messages from the two CPU cores.

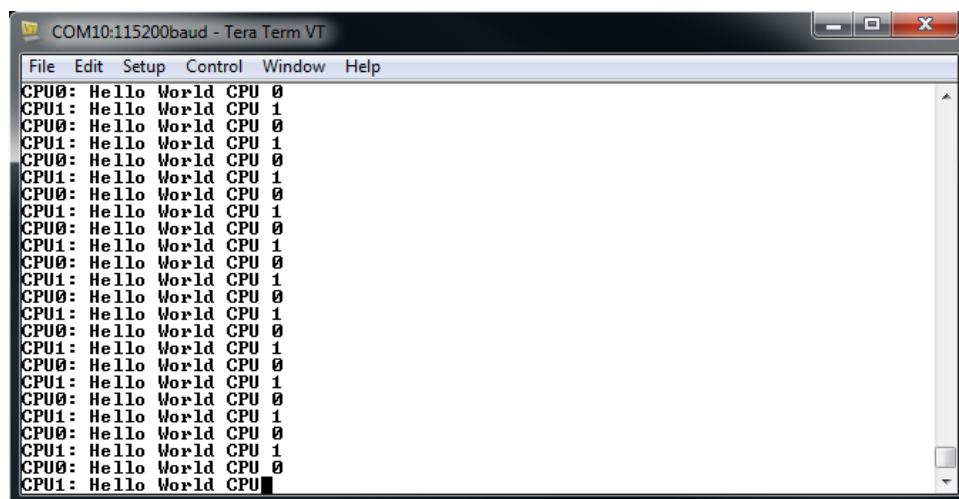


Figure 182 – Terminal Output from CPU0 and CPU1 Applications

2. Connect the JTAG as follows:

- a. ZedBoard – Connect a 2nd micro-USB cable between the host machine and connector J17 (JTAG)
- b. MicroZed -- Connect a Platform Cable or Digilent Programming cable from the host machine to the 2x7 JTAG socket on MicroZed, J3.
- c. PicoZed – Connect a Platform Cable or Digilent Programming cable from the host machine to the 2x7 JTAG socket on PicoZed, J7.

For MicroZed & PicoZed, the recommended Digilent **HS3 USB-JTAG** cable will be used however the Xilinx Platform Cable USB II should work in a similar fashion.

3. In SDK, a new application debug configuration must be created. Debug configurations associate an ELF object file to a target for execution and establish a connection with a debug “server” to perform application debug exercises.

In the **Project Explorer** tab, right-click the **app_cpu0** project and select the **Debug As→Debug Configurations...** menu item.

4. From the Create, manage, and run configurations dialog box, double-click the **Xilinx C/C++ application (System Debugger)** option to create a new TCF debug configuration named **app_cpu0 Debug** by default.
5. Click on the new TCF debug configuration **Application** tab to select it.

Leave the **Download application** option selected in order to load the current application executable into the CPU0 memory so that we can be sure that we are debugging the correct application executable against the current source code. Since each of the CPU cores are already booted by the FSBL, there is no need to use the **Reset processor** option. Also, we should suspend execution by using an implicit breakpoint at the **main()** entry to make it easy to observe the CPU signaling steps at startup.

Deselect the **Reset processor** checkbox option and select the **Stop at program entry** checkbox option.

Verify that **app_cpu0** is shown as the **Project Name** and that the Application has the **Debug/app_cpu0.elf** entry.

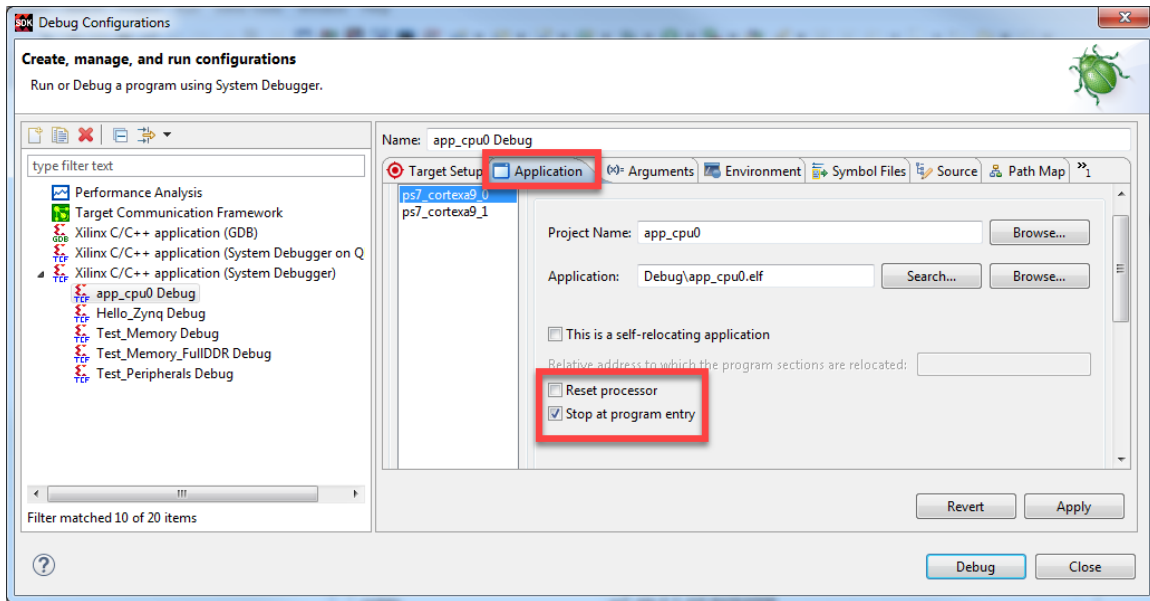


Figure 193 – Processor ps7_cortexa9_0 Application Configuration

6. In the same Application tab under the **Available processors** list, select the **ps7_cortexa9_1** entry and click the **Download application** checkbox option.

This enables a second application to be specified for CPU1 execution and debug. Deselect the **Reset processor** checkbox option and select the **Stop at program entry** checkbox option.

IMPORTANT NOTE: On some smaller resolution screens, the **Debug Configurations** window default sizing does not allow all of these settings to be displayed at once which also can prevent some of the fields from being seen without first either expanding the size of this window or scrolling up within the **Application** tab in order to make the **Download application** checkbox visible on screen.

Verify that **app_cpu1** is shown as the **Project Name** and that the Application has the **Debug/app_cpu1.elf** entry.

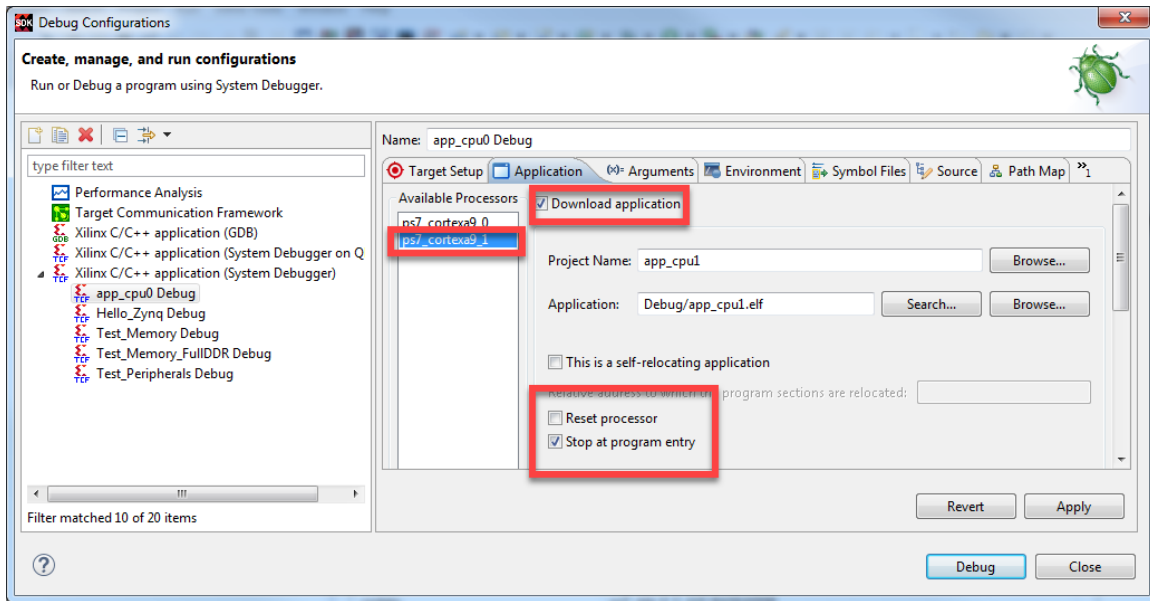


Figure 204 – Processor ps7_cortexa9_1 Application Configuration

- Click on the new TCF debug configuration **Target Setup** tab to select it.

Since each of the CPU cores are already booted by the FSBL, there is no need to use the **Run ps7_init** or **Run ps7_post_config** options since this will only re-initialize the CPUs and could potentially undo or alter some of the work already performed by the FSBL.

Deselect the **Run ps7_init** and **Run ps7_post_config** checkbox options.

Click the **Apply** button then the **Debug** button to launch the TCF debug session.

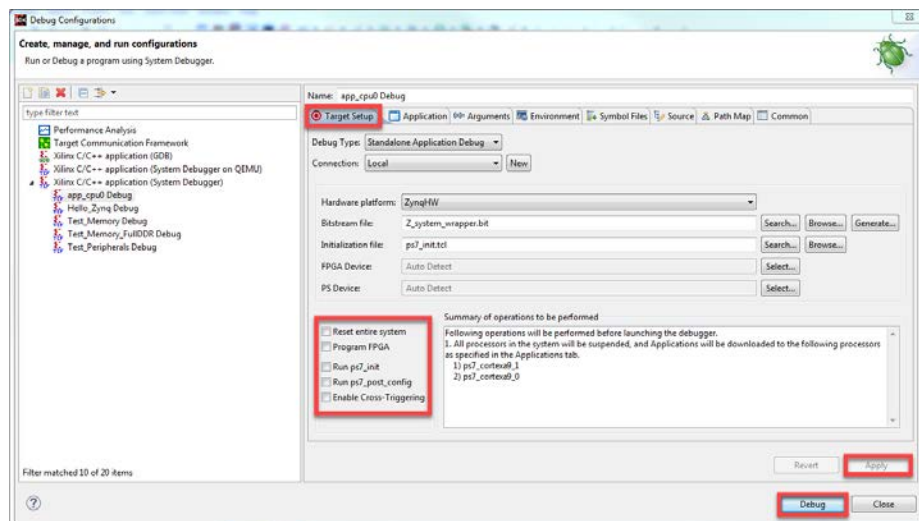


Figure 215 – Target Setup

8. SDK will prompt you to confirm the switch from the **C/C++** development perspective to the **Debug** perspective. Click the **Yes** button to continue launching the debug session.

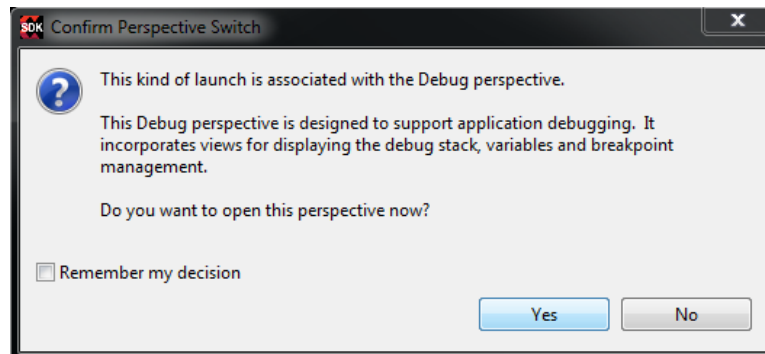


Figure 226 – Switch SDK Perspectives

9. In the TCF **Debug** session window **app_cpu0 Debug** will be shown and the **APU** hierarchy can be expanded to show application execution on each **ARM Cortex-A9 MPCore**.

The applications on core 0 and core 1 should be suspended upon connection. This can be confirmed by checking the terminal output in Tera Term to see that the **print()** messages are no longer scrolling by.

Select the **main()** function listing under the core 0 connection and open the corresponding application source code file **app_cpu0/src/app_cpu0.c** by clicking on the stack entry.

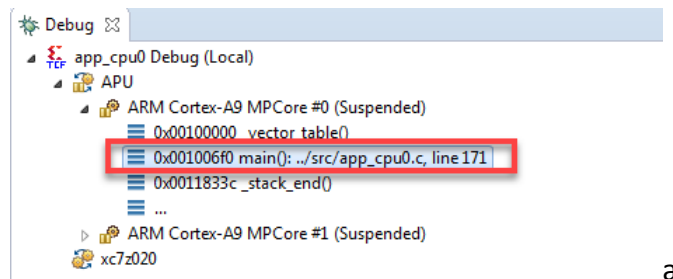


Figure 27 – The app_cpu0.c Stack Entry

10. Set a breakpoint at line **174** of **app_cpu0.c** by double clicking within the hatched blue area to the left of the source code.

This will help us to pause application execution once this core has been signaled to access resources shared between the two cores.

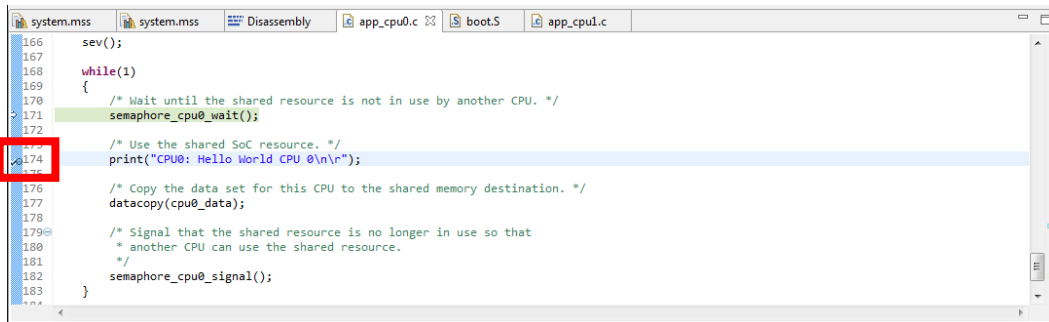


Figure 238 – Breakpoint Set at Line 174 of app_cpu0.c

- Occasionally, core 1 will be reset upon connection and then suspend on boot.S, line 154. If this happens, simply select the core 1 connection in the TCF **Debug** session window and then click the **Resume** button one to advance to the beginning of the **main()** function.

In the TCF **Debug** session window, select the **main()** function listing under the core 1 connection and open the corresponding application source code file **app_cpu0/src/app_cpu1.c** by clicking on the stack entry.

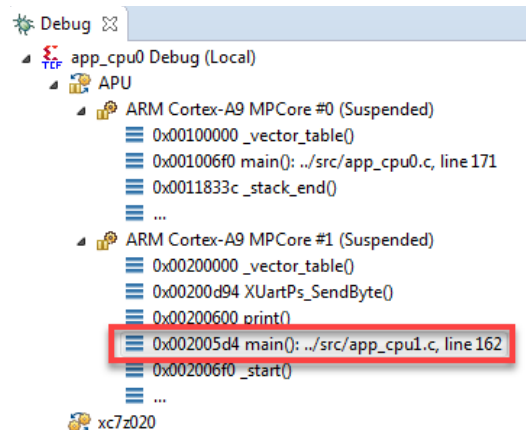


Figure 29 – The app_cpu1.c Stack Entry

- Set a breakpoint at line **162** of **app_cpu1.c** by double clicking within the hatched blue area to the left of the source code.

This will help us to pause application execution once this core has been signaled to access resources shared between the two cores.

```

155 semaphore_cpu1_signal();
156
157 while (1)
158 {
159     /* Wait until the shared resource is not in use by another CPU. */
160     semaphore_cpu1_wait();
161
162     print("CPU1: Hello World CPU 1\n\n");
163
164     /* Copy the data set for this CPU to the shared memory destination. */
165     datacopy(cpu1_data);
166
167     /* Signal that the shared resource is no longer in use so that
168      * another CPU can use the shared resource. */
169     semaphore_cpu1_signal();
170 }
171
172 return 0;

```

Figure 30 – Breakpoint Set at Line 162 of app_cpu1.c

13. Clear the Tera Term buffer so that the application messaging during the startup sequence is not obscured by previous messages. This can be done using the **Edit→Clear buffer** menu option.
14. In the TCF **Debug** session window, select the core 0 connection and then click the **Resume** button to take the application out of the suspended state and advance execution to the implicit breakpoint at the beginning of the **main()** function.

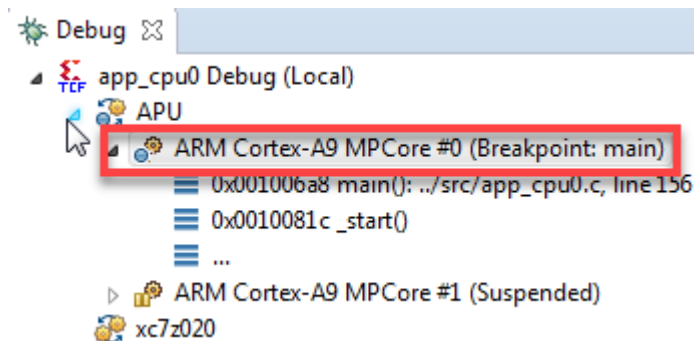


Figure 241 – app_cpu0.c at main() Breakpoint

15. Repeat the actions of Step 14 but this time for the core 1 connection.

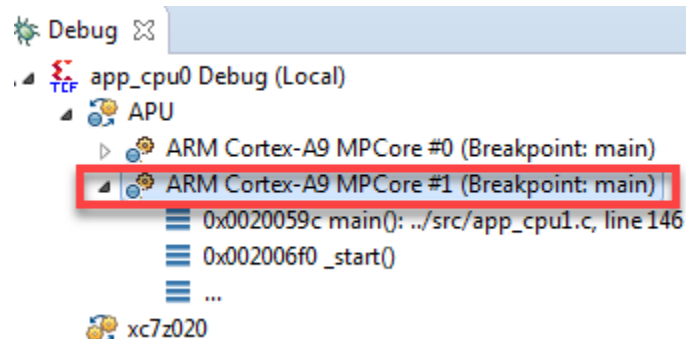


Figure 252 – app_cpu1.c at main() Breakpoint

In the TCF **Debug** session window, select the core 0 connection and then click the **Resume** button again to advance execution to the user breakpoint which was set at line 174 of app_cpu0.c during Step 10 above.

Notice in the serial terminal that CPU1 execution address has been setup by CPU0 and that the SEV signal has been sent as well.

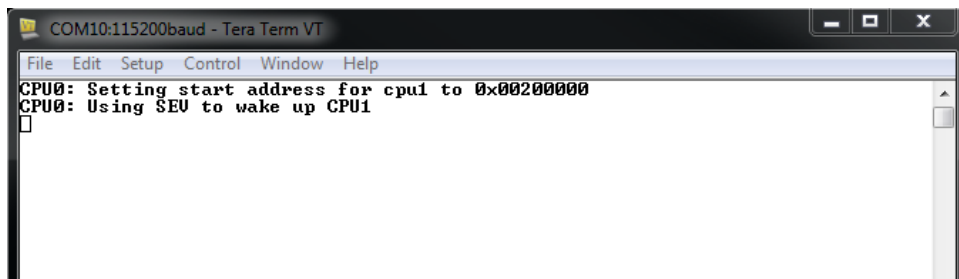


Figure 263 – CPU0 Initializing CPU1 Execution Address

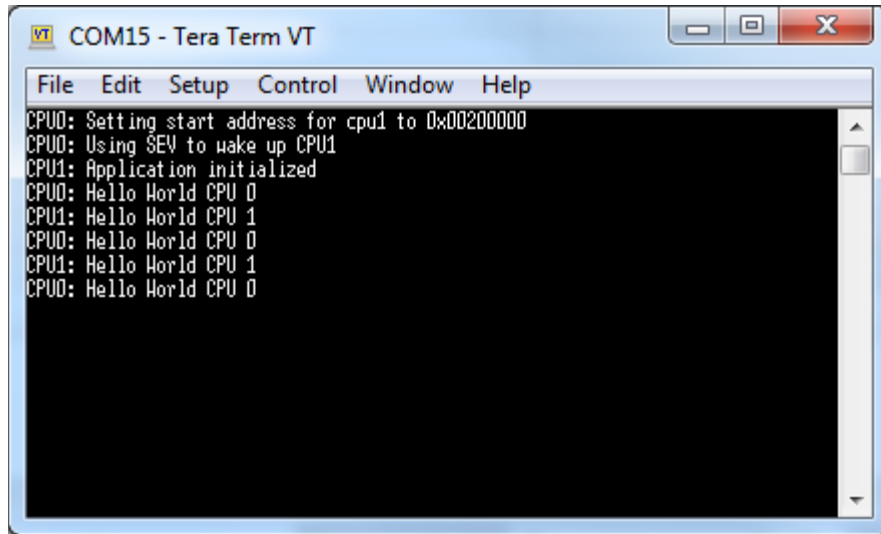
16. Click the **Resume** button once again to start CPU0 into the big **while()** loop and write out the first **Hello World** message. At this point, core 0 has signaled core 1 by writing to the semaphore and will not print a new message until the semaphore is cleared by the core 1 application.

17. In the TCF **Debug** session window, select the core 1 connection and then click the **Resume** button again to advance execution to the user breakpoint which was set at line 162 of app_cpu1.c during Step 12 above.

Notice in the serial terminal that the CPU1 application has been initialized.

18. Click the **Resume** button repeatedly, notice how each core context is switched in the TCF **Debug** session window depending upon which breakpoint gets hit.

Also notice how a **Hello World** message is printed cleanly to the terminal window depending upon which core execution context is being resumed.



```
COM15 - Tera Term VT
File Edit Setup Control Window Help
CPU0: Setting start address for cpu1 to 0x00200000
CPU0: Using SEV to wake up CPU1
CPU1: Application initialized
CPU0: Hello World CPU 0
CPU1: Hello World CPU 1
CPU0: Hello World CPU 0
CPU1: Hello World CPU 1
CPU0: Hello World CPU 0
```

Figure 274 – Terminal Output from Each Application

19. Evaluate what is being performed within the **datacopy()** functions on each CPU.

The location of the definition of this function is not known without digging around in the source code or searching for the string “datacopy”. One shortcut to the function definition is to highlight the **datacopy** function call name, right-click on the text selection, and then click on the **Open Declaration** menu item.

20. In the **datacopy()** function definition, we can see that some byte data is being copied to a memory location at **DESTINATION_ARRAY**. Rather than search through code for the **DESTINATION_ARRAY** definition, simply highlight the definition name, right-click on the text selection, and then click on the Open Declaration menu item.

21. The memory location is again obscured by another definition named **SHARED_DDR_MEMORY_BASE**. Perform the Open Declaration action on this definition as well.

22. The **SHARED_DDR_MEMORY_BASE** definition should be referencing a memory location within the DDR memory space at address **0x03000000**. We can use this address to look into memory to see what sort of data is being copied.

Locate the **Memory** panel within SDK. If the **Memory** panel is not visible within your SDK window, click on the **Window→Show View→Memory** menu item to display it.

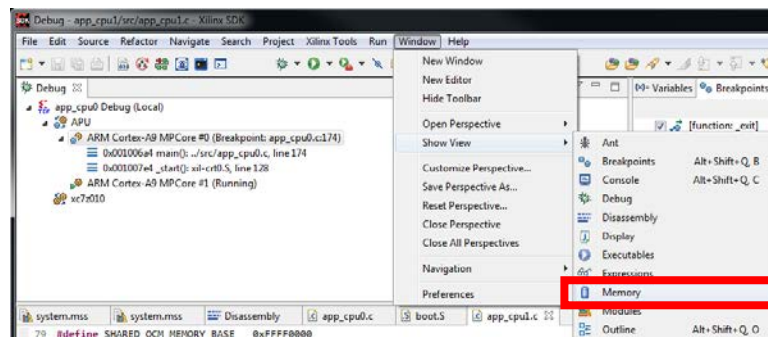


Figure 285 – Displaying the Memory Panel

23. Select the **Memory** panel and click on the Add Memory Monitor button and enter the address **0x03000000** so that the DDR memory contents can be viewed.

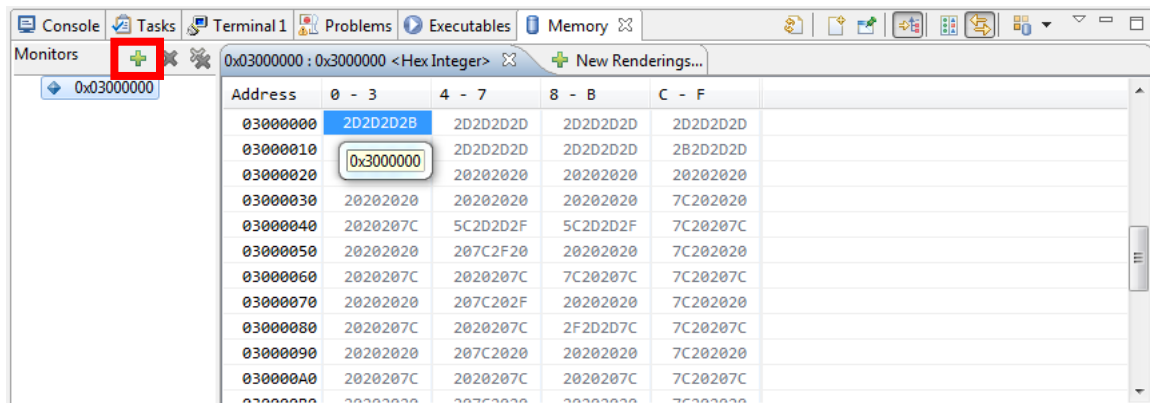


Figure 296 – Hexidecimal Rendered View of 0x03000000 Memory Space

24. This hexadecimal view does not provide us with a lot of information on what is going on in this shared memory space. There are a lot of 0x20 values which correspond to the ASCII white space character so perhaps it would be more useful to view this memory as ASCII characters with a different rendering.

In the **Memory** panel click on the **New Renderings** tab, select the ASCII rendering option, and click the **Add Rendering(s)** button.

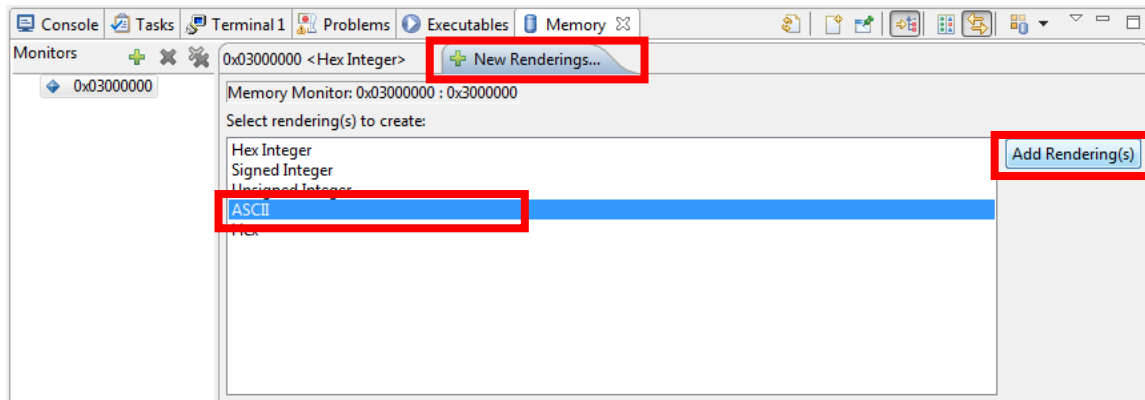


Figure 307 – Creating an ASCII Rendering Memory View

25. Our suspicion of ASCII characters in this memory space appears to be correct, but the contents are not that legible with the current format.

Right click within the ASCII rendered view and select the **Format** option.

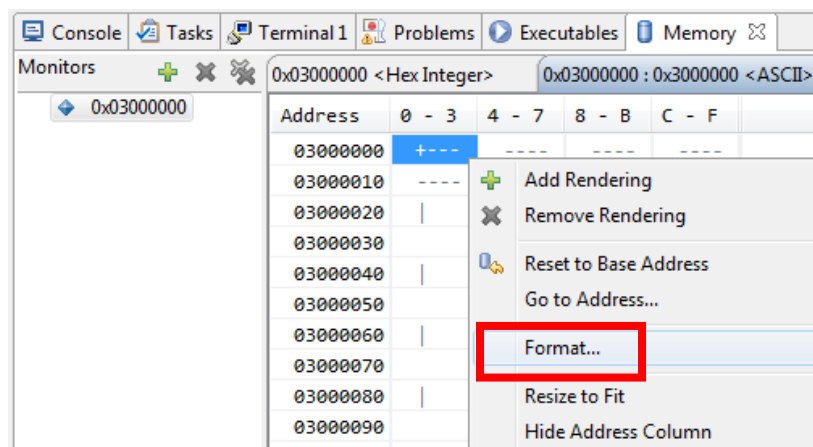


Figure 318– Adjusting the Format of the ASCII Rendered View

26. In the **Format** dialog, set the **Row Size** to **32** and then click the **OK** button.

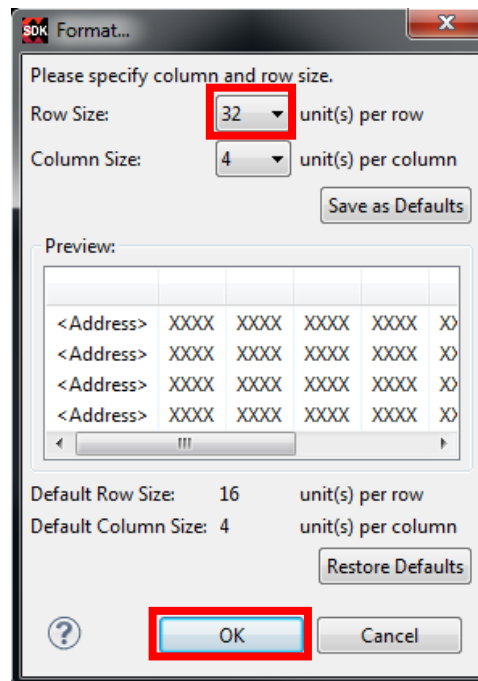


Figure 329 – Setting the Row Size to 32 units per row

27. Adjusting the Format to 32 units per row appears to be effective at making the ASCII text more legible.

Observe the text pattern within this area of memory. The text displayed is the result of the **datacopy()** function from the last CPU to execute it prior to hitting either a breakpoint or the **semaphore_cpuX_wait()** synchronization function.

For the screen shown here, core 0 is stopped at the breakpoint so core 1 executed the **datacopy()** function last.

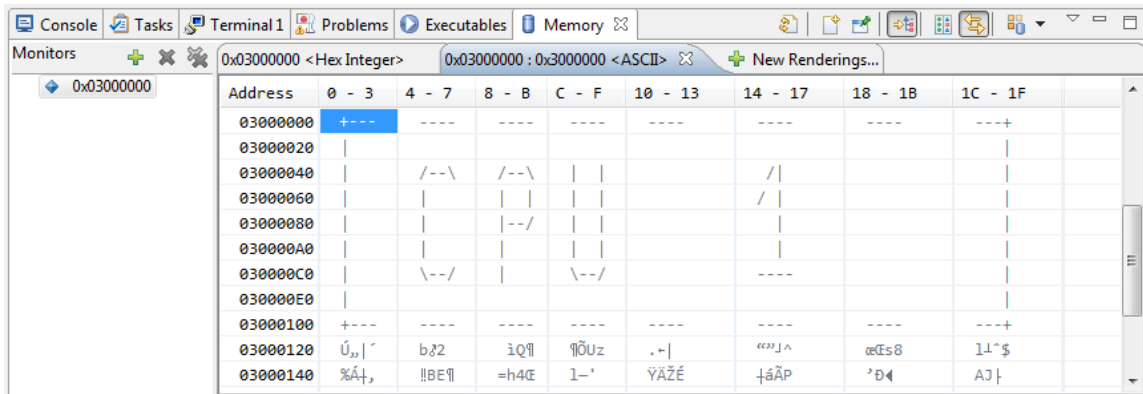
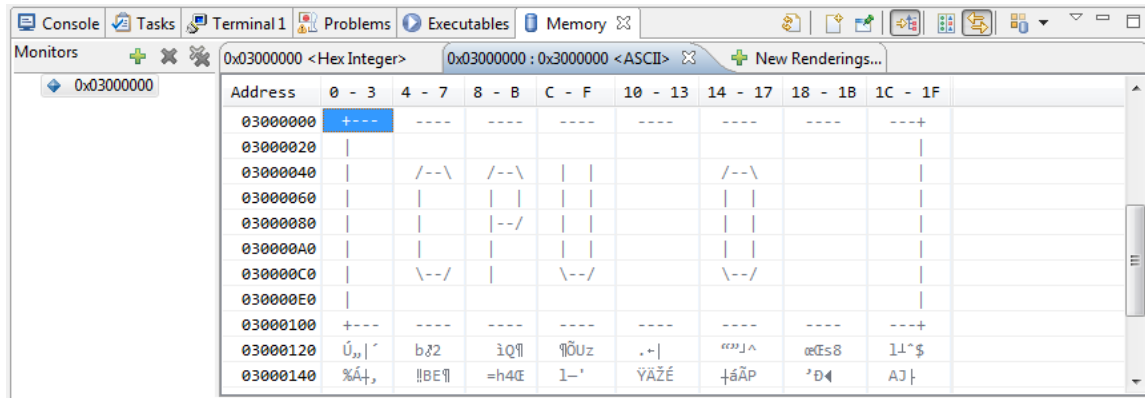


Figure 40 – Adjusting the Format of the ASCII Rendered View

28. Click the **Resume** button once, notice how the memory view blanks when execution context switches to the opposite core after the next breakpoint is reached.

Repeat Steps 24 to 28 above to setup an ASCII rendered view for the other core execution context.

29. Once the ASCII rendered memory view for the other core execution context has been setup, the **datacopy()** results from the other core should be visible.



Address	0 - 3	4 - 7	8 - B	C - F	10 - 13	14 - 17	18 - 1B	1C - 1F
03000000	+	---	---	---	---	---	---	---
03000020								
03000040		/--\	/--\			/--\		
03000060								
03000080								
030000A0								
030000C0		\--/		\--/		\--/		
030000E0								
03000100	+	---	---	---	---	---	---	---
03000120	Ú, '	b¿2	ìQ¶	¶ÖUz	.+	ααJ^	αEs8	11^\$
03000140	%Ä+	!!BE¶	=h4E	1-'	YÄZÉ	+äÄP	'D¶	AJ

Figure 41 – Format of the ASCII Rendered View for the Other Core

30. Click the **Resume** button repeatedly, notice how each core context is switched in the TCF **Debug** session window depending upon which breakpoint gets hit.

Also notice how the contents of the ASCII rendered memory view change after each time the breakpoint is reached.

31. With execution paused at one of the breakpoints on one core, click on the other core's execution context and watch the memory view. Does the content of the memory view display what you expected?

If you pause execution of the running core by clicking the **Suspend** button, does the content of the memory view now display what you expected?

Notice the highlighted changes that are also marked with the delta Δ symbol.

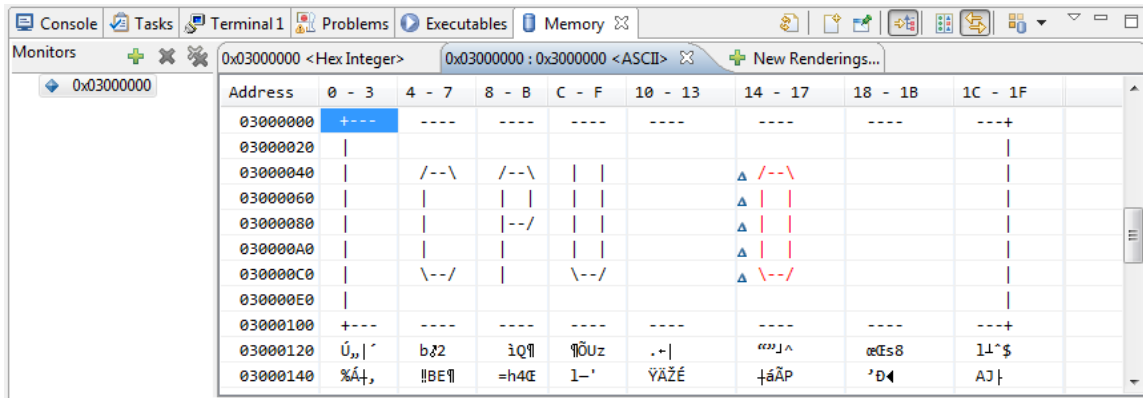


Figure 332 – Format of the ASCII Rendered View after Suspending the Other Core

32. When you are finished with the debug session, click the **Disconnect** button to terminate the TCF debug session.

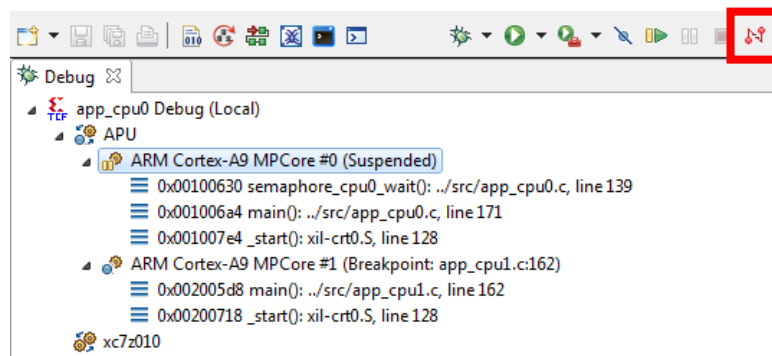


Figure 34 – Disconnecting the TCF Debug Session

Questions:

Answer the following questions:

- Did the memory view contents display what you expected when execution is paused at one of the breakpoints on one core but you are viewing the execution context of the other core which is still running?

Exploring Further

If you have more time and would like to investigate more...

- Remove the semaphore calls to **semaphore_cpuX_wait()** and **semaphore_cpuX_signal()** surrounding the usage of shared resources such as DDR memory and UART. Re-run the applications and discover what application behavior changes.
- Look up the TLB attribute assignments set by the call to **Xil_SetTlbAttributes()** in UG585 - Zynq Technical Reference Manual.

This concludes Lab 9.

Revision History

Date	Version	Revision
12 Nov 13	01	Initial release
22 Nov 13	02	Revisions after pilot
01 May 14	03	Correction to memory space in Exercise 2, Step 12 ZedBoard.org Training Course Release
14 Dec 14	04	Review for update to Vivado 2014.3
07 Jan 15	05	Update to 2014.4
09 Mar 15	06	Merge MicroZed and ZedBoard instructions
18 Mar 15	07	Finalize SDK 2014.4
Oct 15	08	Update to SDK 2015.2
Aug 16	09	Updated to SDK 2016.2

Resources

www.microzed.org

www.picozed.org

www.zedboard.org

www.xilinx.com/zyng

www.xilinx.com/sdk

www.xilinx.com/vivado

www.xilinx.com/support/documentation/sw_manuals/ug949-vivado-design-methodology.pdf

www.xilinx.com/support/documentation/sw_manuals/ug1046-ultrafast-design-methodology-guide.pdf

Answers

Experiment 1

- *What is the difference between a normal standalone application running on CPU0 and a standalone application running on CPU1?*

There is not necessarily a difference between these two applications other than the necessity of the CPU0 application to manage access to resources shared with CPU1. This can be done through either resource arbitration (such as the DDR) or through inter process signaling (such as a semaphore) in order to avoid contention between the CPUs.

Experiment 2

- *Why are two separate application projects required for this applications?*

Since Zynq shares the DDR memory space between the two CPU cores, each application needs to have its own memory space defined in separate linker scripts and built into the executable at link time.

Experiment 3

- *What defines the order in which the applications must be loaded to each of the processor cores?*

This is a function of the both the Zynq BootROM and the FSBL. The FSBL is an SDK application for which the Xilinx source code is provided. Since you have access to the source code, you could modify the FSBL to alter the application load order if you should desire this behavior for your Zynq end application.

Experiment 4

- *Did the memory view contents display what you expected when execution is paused at one of the breakpoints on one core but you are viewing the execution context of the other core which is still running?*

Execution context data, such as memory and register content cannot be updated on the fly with typical debugger technology such TCF. The CPU must be suspended either with a Suspend JTAG instruction or paused at a breakpoint in order for the context information to be updated within the debug session.